Lewis GRANT
IN-61-CR
317403
11P

A VERY HIGH LEVEL LANGUAGE FOR LARGE-GRAINED DATA FLOW*

Hilda M. Standley
Assistant Professor

Dept. of Computer Science and Engineering
The University of Toledo
Toledo, OH 43606

September 5, 1986

ABSTRACT

A data flow language above the level of a traditional high level language is presented for the purpose of adapting programs written in a conventional programming language to a parallel environment. A program written in EASY-FLOW is a set of subprogram calls as units, structured by iteration, branching, and distribution constructs. The sequencing of execution within these structures is dictated by the data dependencies between units.

INTRODUCTION

Recognition of the increases in computation speed offered by a multiplicity of processing elements, accompanied by the steady decline of the cost of a single processing element, has prompted the design of many multiprocessor architectures. The problems of synchronization and communication between multiple processing elements have been addressed satisfactorily to the point that several commercial multiprocessor systems are available at the advertised level of the "minisupercomputer" and "personal supercomputer" [9,12]. The major problems that remain relate to programming the multiprocessors efficiently [7]. A solution to the parallel software problem must provide for: (1) the determination of potential parallelism in programs, (2) partitioning the programs, and (3) scheduling the program partitions to execute in a cooperative fashion.

* This work is supported by NASA Lewis Research Center under grant # NAG 3-699.

DATA FLOW APPROACH TO PROGRAMMING

The data flow schema of program execution offers a simple solution to the software problem [2]. The data flow model of a computation consists of a graph in which nodes represent operations and directed edges represent data dependency relationships. Tokens representing data values travel on the edges, each between the node from which it was output and the node to which it is input. A node is enabled for execution when a token has arrived on each of its input edges. Performing the operation consists of consuming the input token on each input edge and producing output tokens, one on each of the outgoing edges, as a result of the computation. The computations associated with the nodes are assumed to be low-level, binary operations.

The sequence of node executions is determined solely by data availability. No program counter is required. Nodes that are not connected by paths in the graph may execute overlapping in time. Concurrency is determined by default, i.e. by the lack of data dependencies between any two nodes. All potential concurrency, down to the operation level, is exposed. Due to the functional relationship between a result produced by an operation and the corresponding operands,the data flow model may be considered to belong to the class of functional programming models [10], with all of the attendant advantages.

The data flow model of com[illegible] offers assistance in meeting all three software requirements for multiprocessor machines. The maximum amount of potential parallelism is expressed in the data flow graph, the program partitioning problem becomes a graph partitioning problem, and the flow of data provides the automatic scheduling of operations.

Despite the attractiveness of the data flow approach to parallel programming, the implementation on many contemporary multiprocessor architectures poses a major problem. Ironically, it is due to an acknowledged major asset of data flow--the vast amount of parallelism exposed. The large number of potentially parallel operations can cause a large amount of overhead due to the communicating of operands/results between processing elements. Researchers encountering this problem have resorted to the partitioning and compaction of portions of a data flow graph into sequentially executable modules to be run on one processing element [6,13]. The complexity of the task of determining optimal or near optimal partitionings has been documented [14]. This top-down, then bottom up, two step procedure--determining the data dependencies at the operation level and then combining operations into sequential modules--represents a large expenditure of effort. This can be alleviated by the large-grained data flow approach [4] in which the data dependencies are determined at the subprogram level instead of at the operation level.

DATA FLOW PROGRAMMING

The two major approaches to the determination of data dependencies are to use existing, sequential languages, or to develop a new language for the expression of data dependencies. The former approach requires a parallelizing compiler to automatically detect the data dependencies between operations. The obvious advantages are that the retraining of conventional language programmers would be unnecessary and the magnitudes of existing sequential software would still be usable. Much research effort has been expended in this direction with some encouraging results, but many conventional language constructs defy accurate analysis for data dependency information [7].

A parallel language may be designed to include explicit parallel constructs so that the programmer is conscious of the parallelism being expressed, for example: vector operations. In contrast, a new language may provide for the implicit expression of parallelism, by the absence of data dependencies. Data flow languages such as VAL [11], ID [3], and LUCID[15] use this approach.

A VERY HIGH-LEVEL DATA FLOW LANGUAGE

The object of this report, the very high-level data flow programming language, EASY-FLOW, is the result of applying a composite of these approaches to the development of parallel software. The goals of the language design project are: (1) to develop a language that would require very little retraining of conventional language programmers, (2) to provide for the continued use of the magnitude of software in existence with only minor modifications, and (3) to expose potential parallelism both implicitly and explicitly, at the large-grained level or below (referred to as "variable resolution [8]").

Using EASY-FLOW, a program is specified as a hierarchy of units. Each unit may be made up of a substructure of units, a reference to an external unit, or be atomic. An atomic unit is a call to a subprogram, procedure, or function expressed in a conventional, high-level language. It is assumed that the high-level language of the atomic units will be one of the more traditional high-level languages such as FORTRAN, Pascal, or C.

EASY-FLOW defines a unit by specifying subunits (if any) and the relations between them. The relationships between units are determined from the data dependencies deduced from the lists of "input values" and "output values" associated with each unit (the input and output lists associated with the highest level unit, i.e. the program unit, indicate values used in input/output operations with the external environment). The "single assignment rule[1]," important in determining true data dependencies, is enforced with each name representing a unique value or data structure. As a consequence of this rule, no name may appear in both the "input" list and the "output" list of any given unit or in more than one output list, with the exception of the branching construct. "Reassignment" is allowed only as part of the iteration construct.

EASY-FLOW allows the specification of functional/data flow programming in a procedural context. Three language constructs offer the minimal set required to provide for the flow of control: sequence, branch, loop [5]. They are the SUBPROGRAM call, the IF-THEN-ELSE, and the ITER (for iterate), respectively. An additional construct, DISTRIBUTE, offers an explicit notation for parallelism. The calculation of values and the assignment of those values to variables may take place only in a SUBPROGRAM call.

The single assignment rule is enforced, as previously stated, by not allowing the same name to appear in both the input list and the output list of the same unit. In addition to this, each call to a subprogram is enclosed within an envelope (the INTO-OUTOF pair) that allows the passing of parameters while shielding the variable in the unit from reassignment through returning parameter values. Global variables are prohibited since their values may be changed as the result of subprogram calls.

TAKING ADVANTAGE OF PARALLELISM

The unit, IF-THEN-ELSE, ITER, and DISTRIBUTE constructs provide a structure within which one or more units may be placed. Multiple units appearing within a structure are termed a unit set. A data flow graph is constructed by the EASY-FLOW language processor from the data dependencies determined from the input and output lists of the units within a unit set. An edge between two units is established wherever a data value name appears in the input list of one unit and the output list of the other. Implicit parallelism may be found by examining the data flow graph. The nodes in the graph that are not connected by paths may be scheduled to be executed overlapping in time.

The example EASY-FLOW program in Figure 1 calculates the maximum of the values of the two functions, f and g at a point X. The program is named MAX_TWO and four names for data values are declared having type real. The program consists of one unit, called MAIN, with one input value, X, and one output value, RESULT. The body of the MAIN unit is a unit set consisting of three units, each having a call to a subprogram as its body, for calculating the functions f(X), g(X), and max(f(X),g(X)). Since the stated data dependencies between units determine the sequence of execution, these three units may appear in any order. The data flow graph representing this calculation is shown in Figure 2. Potential parallelism is seen to exist between the calculations of f(X) and g(X). A subprogram written in a conventional language must be provided for each subprogram called from the EASY-FLOW program. In this example three subprograms are required: F, G, and MAX in order to calculate f(x), g(x), and max(f(x),g(x)), respectively.

Figure 3 gives an EASY-FLOW program for performing the calculation of MAXTWO on several values of X. The ITER statement provides for the looping and changing of the value of the index, I, over the array X. As a way of introducing the IF-THEN-ELSE statement, a zero value is produced as the result of an X value less than zero. Subprograms B1 and B2 must be provided in a conventional language to return boolean values as the result of tests for I less than or equal to 10 and X(I) greater than or equal to 0.0, respectively. The program in Figure 4, MANYMAXTWOPAR, performs the same calculation, making use of the distribute construct to explicitly state the parallelism in independently processing all elements of the X array.

PROGRAMMING IN EASY-FLOW

EASY-FLOW may be used for writing new programs or for modifying existing programs for the purpose of producing parallel software. The hierarchical nature of the language assists in top-down program design. Each unit may have a substructure consisting of a unit set. At the lowest level, subprograms written in a conventional programming language must be supplied. These subprograms will perform all programming operations not included in EASY-FLOW and will typically include assignment statements, arithmetic expressions, boolean expressions, and operations on data structures. Since data dependencies between units are obstacles to parallelism, designing separate units to operate on data separately promotes parallelism.

Existing programs may be modified in a top down fashion with the main program being rewritten in EASY-FLOW as a set of calls to the subprograms. Additional subprograms at levels below the main program may be rewritten into EASY-FLOW with the degree to which this is done determining the amounts of parallelism to be made available, i.e. only the parts of the program written in EASY-FLOW will offer parallelism; all other parts will remain to be executed sequentially. Thus, the amount of parallelism exposed is proportional to the amount of effort expended in rewriting in EASY-FLOW.

The EASY-FLOW approach to large-grained data flow programming differs from previous efforts [4] primarily by the absence of the requirement that the programmer adhere to a graph model. Programs are specified in the standard textual fashion and, with the exception of the DISTRIBUTE construct, do not indicate any explicit parallelism.

THE EASY-FLOW LANGUAGE PROCESSING SYSTEM

The major task of a compiler for EASY-FLOW is to determine the data flow graph for each unit set encountered in a program. The code produced by a compiler for EASY-FLOW is dependent upon the parallel nature of the target machine. In the case of a uniprocessor, completely sequential code is produced for the entire program. This may be done by performing a topological sort on the nodes in the data flow graph to determine an appropriate, although not necessarily unique, unit execution sequence.

In the case of a multiprocessor system, the data flow graphs generated offer a guide to the separation and distribution of units. Once program units have been assigned to processing elements, execution may begin with the communication of data values between units by whatever means available on the target machine. One of the parallel architectures being examined in this project is the Intel iPSC** Concurrent Computer. Current operating system software supports individual high-level language programs on each processing element, with communications between programs taking place through message passing. Program units assigned to individual processing elements are augmented with statements to facilitate the receipt of input data (operands) and to cause the results to be passed along as outgoing messages.

Additional software exists to check the subprograms written in the conventional languages for potential violations to the single assignment rule. Subprograms are processed by a "sanitizing" program to remove references to global variables and to make note of name equivalencing statements (examples: COMMON and EQUIVALENCE statements in FORTRAN, respectively).

CONCLUSIONS

A new data flow language, EASY-FLOW, has been presented with the main goal of easing the transition from traditional high-level language programming to programming in languages that promote the production of parallel software for contemporary multiprocessor computers. Although the concept is (perhaps deceptively) attractive in its simplicity, it remains to be seen if it is genuinely easy for traditional programmers to use and if sufficiently efficient implementations result.

The intent of the language is to offer a simple structure for the determination of parallelism at a high-level, i.e. large-grained parallelism. However, due to its hierarchical nature, any level granularity may be achieved, down to the operation level in one assignment statement. It is not expected that a programmer using EASY-FLOW will carry the granularity down to the point where a subprogram contains only one assignment statement, but that capability exists. Programs specified in a relatively fine-grained fashion may have selected units above the atomic level processed to produce sequential code when the architecture of the target machine dictates that the granularity of the program is too fine to permit an efficient

** iPSC is a registered trademark of Intel Corporation.

implementation. The automatic determination of optimal program granularity is a useful language processing system extension and continues to be an open area of research.

ACKNOWLEDGEMENT

The following people are gratefully acknowledged for their important contributions to this project: Mary Lou Dorf and Shelly Hancock, both of The University of Toledo, and Jim Kiraly of the NASA Lewis Research Center.

```
MAXTWO:

  declare:  real X,FX,GX,RESULT

  unit MAIN:
    input: X

      unit CALCF:
        into: X => X
        subprogram F(X,FX)
        outof:FX => FX
      endunit CALCF:

      unit CALCG:
        into: X => X
        subprogram G(X,GX)
        outof:GX => GX
      endunit CALCG

      unit FINDMAX:
        into: FX => FX
              GX => GX
        subprogram MAX(FX,GX,RESULT)
        outof:RESULT => RESULT
      endunit FINDMAX

    output: RESULT

  endunit MAIN
```

Fig. 1 EASY-FLOW program to calculate the maximum value of the two functions f and g at point x.

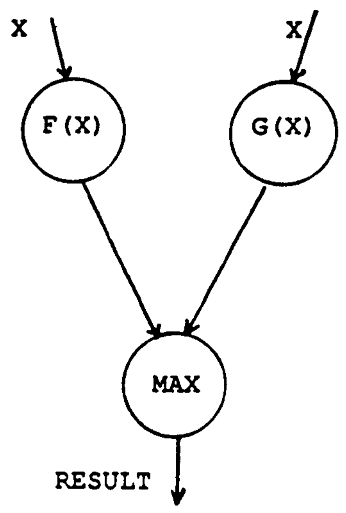

Fig. 2 A data flow graph for MAXTWO.

```
MANYMAXTWO:

  declare:  real X(10),FX,GX,RESULT(10)
            integer I,J
            boolean T1,T2

  unit MAIN:
    input: X
    unit INITIALIZE:
      into:
      subprogram INIT(I)
      outof: I => I
    endunit INITIALIZE
    unit LOOP:
      input I,X
      iter  into: I => I
            subprogram B1(I,T)
            outof:T => T1
      do
        unit CHECK:
          input: X,I
            if  into: X(I) => X
                subprogram B2(X,T)
                outof:T => T2
            then
              unit CALCF:
                into: X(I) => X
                subprogram F(X,FX)
                outof:FX => FX
              endunit CALCF
              unit CALCG:
                into: X(I) => X
                subprogram G(X,GX)
                outof:GX => GX
              endunit CALCG
              unit FINDMAX:
                into: FX => FX
                      GX => GX
                       I => I
                subprogram MAX(FX,GX,I,RESULT)
                outof:RESULT => RESULT
              endunit FINDMAX
            else
              unit ZERO:
                into: I => I
                subprogram ZERO(I,RESULT)
                outof:RESULT => RESULT
              endunit ZERO
            output: RESULT
          endunit CHECK
```

```
            unit INCREMENT:
              into: I => I
              subprogram INCR(I)
              outof:I => J
            endunit INCREMENT
          reassign
            J => I
          output: RESULT
        endunit LOOP
        output: RESULT
      endunit MAIN
```

Fig. 3 EASY-FLOW program to calculate the maximum value of the two functions f and g at 10 positive x points. The value of zero is returned if x is less than or equal to 0.

```
MANYMAXTWOPAR:

  declare:  real X(10),FX,GX,RESULT(10)
            integer I,J
            boolean T
  unit MAIN:
    input: X
      distribute I = 1 .. 10
        unit CHECK:
          input: X,I
            if  into: X(I) => X
                subprogram B(X,T)
                outof:T => T
            then
              unit CALCF:
                into: X(I) => X
                subprogram F(X,FX)
                outof:FX => FX
              endunit CALCF
              unit CALCG:
                into: X(I) => X
                subprogram G(X,GX)
                outof:GX => GX
              endunit CALCG
              unit FINDMAX:
                into: FX => FX
                      GX => GX
                       I => I
                subprogram MAX(FX,GX,I,RESULT)
                outof:RESULT => RESULT
              endunit FINDMAX
            else
              unit ZERO:
                into: I => I
                subprogram ZERO(I,RESULT)
                outof:RESULT => RESULT
              endunit ZERO
        output: RESULT
      endunit CHECK
      output: RESULT
    endunit MAIN
```

Fig. 4 Program MANYMAXTWO with explicit parallelism.

Bibliography

[1] W. Ackerman, "Data Flow Languages," AFIPS Conf. Proc., Vol. 48, 1979, pp. 1087-1095.

[2] T. Agerwala and Arvind, "Data Flow Systems--Guest Editors' Introduction," Computer Vol 15, No. 2, Feb. 1982, pp. 10-13.

[3] Arvind and K. Gostelow, "The U-Interpreter," Computer, Vol. 15, No. 2, Feb. 1982, pp. 42-49.

[4] R. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques," Computer, Vol. 17, No. 7, July 1984, pp. 55-61.

[5] C. Bohm and G. Jacopini, "Flow Diagrams, turing Machines and Languages With Only Two Formation Rules," CACM Vol. 9, No. 5, May 1966 pp. 366-371.

[6] M. Campbell, "Static Allocation for a Data Flow Multiprocessor," Proc. 1985 Int. Conf. on Parallel Processing, pp. 511-517.

[7] D. Gelernter, "Domesticating Parallelism--Guest Editor's Introduction," Computer, Vol. 19, No. 8, pp. 12-16.

[8] J. Gaudiot and M. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," Proc. 4th Int. Conf. Distributed Computer Systems, 1984, pp. 2-9.

[9] N. Hayes "New Systems Offer Near-Supercomputer Performance," Computer, Vol. 19, No. 3, Mar. 1986, pp. 104-107.

[10] P. Henderson, "Functional Programming--Application and Implementation," Prentice-Hall International, 1980.

[11] J. McGraw, "The VAL Language--Description and Analysis," Tech. Report UCRL-83251, Lawrence Livermore Laboratory, University of California, Livermore, CA 94550

[12] N. Mokhoff, "Hypercube Architecture Leads the Way for Commercial Supercomputers in Scientific Applications," Computer Design, Vol. 25, No. 9, May 1, 1986, pp. 28-30.

[13] T. Ravi, "Partitioning and Allocation of Functional Programs for Data Flow Processors," Internal Report UCLA Computer Science Dept., Univ. of Ca., Los Angeles, 1985.

[14] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," 1986 ACM Conference on List and Functional Programming.

[15] W. Wadge and E. Ashcroft, "Lucid, the Dataflow Programming Language," Academic Press, 1985.